Bachelor Thesis

Bidirectional Grammar Transformations

Simon Wegendt

Supervisors:
Yufei Cai
Prof. Dr. Klaus Ostermann

October 21, 2015

Contents

1 Introduction

1.1 Motivation

In many languages it's easy to write a certain kind of parser while others may be less intuitive: In functional languages writing a recursive descent parser is easy while designing a bottom-up parser is more complex all around. A recursive descent parser can't handle left-recursive grammars though. The CYK-parsing algorithm only works on grammars in CNF (chomsky normal form) or similar two-forms [1] and, while every context-free grammar can be transformed to a CNF-grammar which produces the same words, the syntax tree produced by parsing the words differs a lot from the ones of the original grammar. Since syntax trees are often more important than the actual words, as the trees often recursively give meaning to their word, this is a serious problem.
Also often you'd like to have a grammar employing many features while dealing with a more simple syntax tree after parsing; for example you might like to enable both of these types of variable declarations while only dealing with one of them in your syntax tree:

```
int (a, b) = (0, 1);
int c = 0, d = 1;
```

Moreover, many textbooks describe grammar transformations like CNF or left recursion elimination. Implementing these is somewhat complicated and it would be nice to be able to reuse some kind of standard definition for them, also the resulting syntax tree of the starting and the transformed grammar will differ. Therefore finding a more concise, correct way to describe grammar transformations would be nice.
For all of these problems we'll try to find a solution.

1.2 Goal

The Goal of this thesis is to

- describe transformations between context free grammars using a DSL ("Domain-specific language")
- apply such transformations
- describe or infer forwards and backwards transformations between the corresponding syntax trees

In this thesis, the transformation between grammars is done using a pattern matching approach matching single grammar rules exhaustively, building a table holding the match information, and constructing rules using the same patterns. Syntax tree transformations can be inferred in some cases, while in others they have to be written done more or less explicitly (although still prototyped, so they will be instantiated on a per-grammar base).

2 Background information on grammars

This section will briefly establish a common ground on what grammars etc. are.

2.1 Theory

A grammar G is a tuple

$$G = (N, \Sigma, P, S)$$

where

N is a finite set of *NTs(nonterminals)*

Σ is a finite set of *terminals*, Σ being disjoint from N

P is a finite set of production rules, each rule of the form

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

$S \in N$ is the *start symbol* of the grammar

2.2 Kleene-operator

Given a set M, M^* is defined as all concatenations of any number (even none) of all elements of M. A more formal definition of this could be

$$M^* = \bigcup_{n=0}^{\infty} M^n$$

In language terms, these tuples are often written without the syntactical sugar, for example to represent a number like 1523 over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, you could use the tuple $(1, 5, 2, 3) \in \Sigma^4 \subseteq \Sigma^*$. Since the tuple representation is not very readable, we'll write 1523 in most cases instead. The empty tuple is often written as ϵ or λ.

2.3 Languages

A *language* is a subset of all words over an alphabet Σ.
Examples: $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- N is equal to Σ^*

- The set of all binary coded numbers is a subset of Σ^*

2.3.1 Languages produced from grammars

Grammars can be used to define a language. Given a grammar $G = (N, \Sigma, P, S)$, the set of all it's words $L(G)$ can be described as all words w in Σ^* where a sequence of derivations exists, such that $S \implies {}^*_G w$.

2.4 Syntax trees

Syntax trees describe how words are formed from grammars. The parent node always contains the left hand side of a production rule, it's children nodes joined together are the corresponding right hand side.
The root node is the start symbol, the leafs form the produced word.

2.5 Context free grammars

Context free grammars are grammars with production rules being limited to only one symbol on the left hand side, therefore every rule has to look like

$$N \to (\Sigma \cup N)^*$$

Context free grammars are much easier handled than those without this limitation, while still being powerful enough to describe the majority of a programming language and most other needed stuff like braced terms etc.

2.5.1 .grammar-file specification

To specify grammars for our program we use custom, simple `.grammar`-files. Here's an example to start:

```
start C
C -> C_1 S "+" C | C_2 S
S -> S_3 F "*" S | S_4 F
F -> F_5 "(" C ")" | F_6 <int>
```

We'll be using this grammar quite often since it describes arithmetic expressions without using left-recursion and with automatic precedence of "*" over "+".
In general, given a context-free grammar $G = (N, \Sigma, P, S)$, it is encoded like this:
The starting symbol S ist to be placed on the first line after the keyword `start`, for example `start S`.
Each rule in P is encoded as follows: We first assign each rule a unique name, allowing us to not only parse, but track exactly which rule was used in parsing and building the syntax tree. We call the rule's type together with it's name its constructor, since, when parsing, it constructs a tree; similar to a data constructor in Haskell. Then rules are encoded like this:
`Nonterminal -> Rulename Atoms`
where

- `Nonterminal` is any alphanumeric sequence beginning with an uppercase letter - this represents the rule type
- `Rulename` is the same nonterminal followed by an underscore and a unique name
- `Atoms` are one or more `Nonterminals` or `Terminals`
- a `Terminal` is any string enclosed by double quotation marks, or a special numeric terminal for convenience: `<int>` or `<float>`.

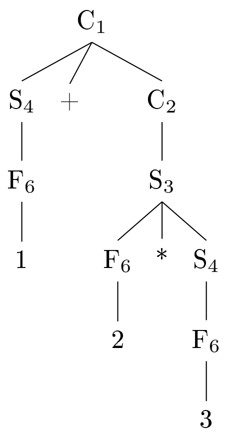
As a shortcut to writing multiple rules with the same type you can write the following:
`Nonterminal -> Rulename Atoms | ... | Rulename Atoms`
You may freely use whitespace in most places, this includes linebreaks before `|`. You may also comment your grammar with `C`-style line comments, i.e. `//comment`.
Both the sets N and Σ are defined implicitly.

3 The DSL

This section will describe how to specify transformations on grammars and syntax trees. We'll first look at how to come up with the patterns for matching and producing grammar rules, and later look at transforming the syntax trees belonging to the grammars. For this, we'll we'll look at the `concrete.grammar` and it's syntax trees:

```
start C
C -> C_1 S "+" C | C_2 S
S -> S_3 F "*" S | S_4 F
F -> F_5 "(" C ")" | F_6 <int>
```

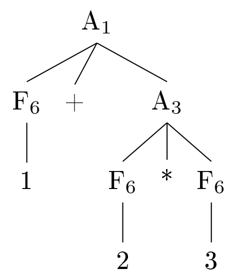
C stands for "concrete", *S* for summand, *F* for factor. The grammar is "concrete" in the sense that it tells you what each subterm is by its type. It also defines a precedence overridable with braces: in the corresponding syntax tree of an arithmetic expression produced by this grammar, evaluating a branch by evaluating each child first, replacing the children by their value and then evaluating the branch itself corresponds to the standard evaluation order of the expression itself. The syntax tree of "$1 + 2 * 3$" for example is:



For most purposes, after parsing we no longer care about the type of the nodes themselves, the tree's shape gives us enough information to evaluate the expression. Therefore it would be enough to have some equivalent syntax tree produced by the following grammar:

```
A -> A_1 A "+" A
   | A_3 A "*" A
   | A_6 <int>
```

The corresponding abstract syntax tree of "$1 + 2 * 3$" with $*$ having precedence over $+$ is:



The abstract grammar, which only loosely describes how arithmetic expressions are formed, does not define a precedence: it's represented by the tree. The concrete grammar is non-ambiguous though: it defines a precedence and allows braces to break it. Transforming the tree forwards and backwards not only keeps this precedence, it eliminates (or adds) unneeded braces. All versions of valid backtransformed trees are obtainable; for convenience the least deep, shortest unparsed is choosen for printing. The forward transformation only produces one tree.
We want all nodes to have the same type, A (for "abstract"). We'd like to eliminate chains of just As as well.

3.1 Grammar transformation

To describe grammar transformations we're using a pattern matching approach. This means that the DSL describes input patterns to match on rules and output patterns to produce new rules. We'll first go through the thougt process of creating those patterns and some other declarations, and later put them together into a complete object.
To match the all rules

```
C -> C_1 S "+" C | C_2 S
S -> S_3 F "*" S | S_4 F
```

we can use the following matcher:

```
Y -> Y_1 Z x Y | Y_2 Z
```

When matching the first two rules, we'll get the following mapping of MatcherAtoms to GrammarAtoms:

$$Y \to S,\ Z \to C,\ x \to "+"$$

while matching the second two rules, we'll get a similar mapping:

$$Y \to F,\ Z \to S,\ x \to "*"$$

Since we want to always get the same symbol, we'll first have to map a NonterminalMatcher to it, which is done using the following declaration:

```
W = A
```

Then to produce the first two abstract grammar's rules, we use the following producer:

```
W -> W_1 W x W
```

With the information from matching and our declaration, this produces the following rule after matching the first two concrete rules:

```
A -> A_1 A "+" A
```

So what is most important about this part of the matching is the information mapped by x and the form of the matched rule. Note, that Y ->Y_1 Z x Y does not match S -> S_1 F "*"S, since x only matches terminals, while Y/ Z only matches nonterminals (and that's what we want). To keep the last rule, we use the following matcher and producer:

```
in
  S -> S_1 x B y | S_2 z:1
out
  W -> W_2 z:1
```

The structured format containing all this information then looks like this:

```
1  start A
2  begin
3    W = A
4    begin
5      W_1 = collapse W C_1
6      in
7        Y -> Y_1 Z x Y | Y_2 Z
8      seq
9        W -> W_1 W x W
10     pattern auto
11       C_2 x = (x :: W)
12   end
13   begin
14     in
15       S -> S_1 x B y | S_2 z:1
16     out
17       W -> W_2 z:1
18     pattern auto
19       S_1 x (y :: B) z = (y :: W)
20   end
21 end
```

Let's go through it part by part:

Line 1: The first line specifies the starting symbol for the produced grammar, so the produced grammar will have the nonterminal A as it's starting symbol.

Lines 2 and 21: **begin**...**end** specifies the begining and and end of a block. A block is applied exhaustively to the input grammar, matching grammar rules and producing new ones. All definitions/bindings/... of a block are scoped and can be overriden by subblocks.

Line 3: Here's the assignment W = A. In detail it means: assign the NonterminalMatcher W the nonterminal A. This is a constant "A" and not one resolved by matching etc.

Line 5: W_1 is a rule name. Here it get's bound to the result of the collapse-function, which evaluates:

- the NonterminalMatcher W
- the rule name assignment C_1
- takes the matched nonterminal from the first step and replaces the nonterminal retrieved from the second step by it

For example, if W matched Abs and C_1 matched Con_sum, W_1 will be bound to Abs_sum.
It's not needed in our example, since the program tries to guess what name to give unknown rules, but if you want to be sure of the mapped name you can use this function to transfer names to new rules.

Lines 6-11: **in** ... **out**/**seq** (... **pattern** (**auto**)(**force**)) is a block of matchers, producers, and patterns.

Lines 15 and 17: Here's our matchers and producers. They will match a grammar rule of the same length, atom by atom.

Line 11: patterns are used to specify complex transformations. This will be discussed in detail later.

In general, the file is defined as follows:
The matcher- and producer-patterns look similar to rules in the .grammar-files:
`TypeVariable -> RuleName MatcherAtoms`
where again

- TypeVariable is any alphanumeric sequence beginning with an uppercase letter
- RuleName is the same type variable followed by an underscore and a unique name. Both can be of arbitrary length but must not contain whitespace. Since the type variable is a nonterminal, it must start with an uppercase letter.
- MatcherAtoms are one or more
  - NonterminalMatchers, an uper-case letter followed by any alphanumeric (e.g. A, Ka, Ze1Z like nonterminals). They match nonterminals.
  - AnyMatchers, an underscore followed by a nonterminal-like (e.g. _A, _Ka, _Ze1Z). They match any grammar atom.
  - TerminalMatchers, like NonterminalMatchers, only starting with a lower-case letter (e.g. a, kA, ze1Z). They match any terminal.
  - LiteralMatchers, any string enclosed by double quotation marks, or a special numeric terminal for convenience: <int> or <float>. They match only their exact grammar counter-parts.

To define a matcher/producer-pair, we enclose it with **begin** and **end** and prefix the matchers with **in**, the producers with **out**.
At the start of a **begin**-block you can declare some things:

- NonterminalMatcher =Nonterminal fixes a nonterminal matcher to the specified nonterminal, which means that in this scope the nonterminal matcher will only match the specified nonterminal.

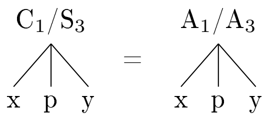
- Rulename =function arguments fixes NTMs or rule name associations whenever the block is run. function is a specifically defined function operating on the arguments. Some are predefined, although you can add your own quite easily.
  - A_1 =collapse A B_1 will for all matched rule names $B_i \to C_j$ and the association $A \to D$ associate $A_i \to D_j$, so if B_1 matched some rule named K_2 and A matched the nonterminal U, A_1 will match U_2.
  - A =newName default or A_1 =newRuleName default will associate the specified matchers with an unused NT or rule name.
- NonterminalMatcher will try to associate a NTM to every NT in the input grammar, but not generate it.

3.2 Syntax tree transformation

To describe how to transform syntax trees we use pattern synonyms. Pattern synonyms describe how (sub-)trees relate to each other. Again we use the concrete and abstract grammar as an example. The pattern synonyms for transforming between them would look like this (with the concrete grammar on the left side and the abstract grammar on the right):

```
C_1 x p y = A_1 x p y
C_2 x     = (x :: A)
S_3 x m y = A_3 x m y
S_4 x     = (x :: A)
F_5 l x r = (x :: A)
F_6 x     = A_6 x
```

The patterns $C_1 \ldots$ and $S_3 \ldots$ are simple: when we encounter a tree constructed by C_1/S_3 we copy the information inside into a tree of type A_1/A_3. It corresponds to the following subtree equivalence:



The lowercase letters in these are variables which will hold some kind of subtree. The patterns $C_2 \ldots$ and $S_4 \ldots$ are a bit more complicated: they state that, when encountering a tree constructed by C_2/S_4, we take it's children and see how we can transform it to a tree of type A matching on this children. We could also write the first one explicitly like this:

```
C_2 (S_3 x m y)                     = A_3 x m y
C_2 (S_4 (F_6 x))                   = A_6 x
C_2 (S_4 (F_5 (C_1 x p y)))         = A_1 x p y
C_2 (S_4 (F_5 (C_2 (S_3 x m y))))   = A_3 x m y
...
```

However, we would only be able to specify transformations with a fixed amount of depth. Therefore, the pattern C_2 x =x :: A is needed.
Since we know that trees are finite, this "implicit"-style will terminate, but work without any depth restriction.
As you can see, patterns may nest like this to access subtrees:

A_1 (B_2 a)b c =...
These patterns can be directly written in the DSL: after declaring the output patterns, write **pattern** followed by your patterns. For more complicated examples see the case studies.
Using all of this, we get the following transformer-file:

```
start A
begin
  W = A
   begin
     W_1 = collapse W C_1
     in
       Y -> Y_1 Z x Y | Y_2 Z
     out
       W -> W_1 W x W
     pattern
       Y_1 x p y = W_1 x p y
       Y_2 x     = (x :: W)
   end
   begin
     in
       S -> S_1 x B y | S_2 z
     out
       W -> W_2 z
     pattern
       S_1 x y z = (y :: W)
       S_2 x     = W_2 x
   end
end
```

The first **begin**/**end**-block will match the rules C_1 to S_4, binding Y to C or S etc. and producing all four pattern synonyms. The second block will match the rules F_5 and F_6, producing the last two patterns.
Pattern can be inferred as well: By specifying how information flows between the syntax trees, pattern can be inferred. To do this you can follow a matcher atom with :ID, for example W:1. The :ID may be any character sequence without whitespace. Then follow the keyword **pattern** with **auto**. In the example, this will look like this:

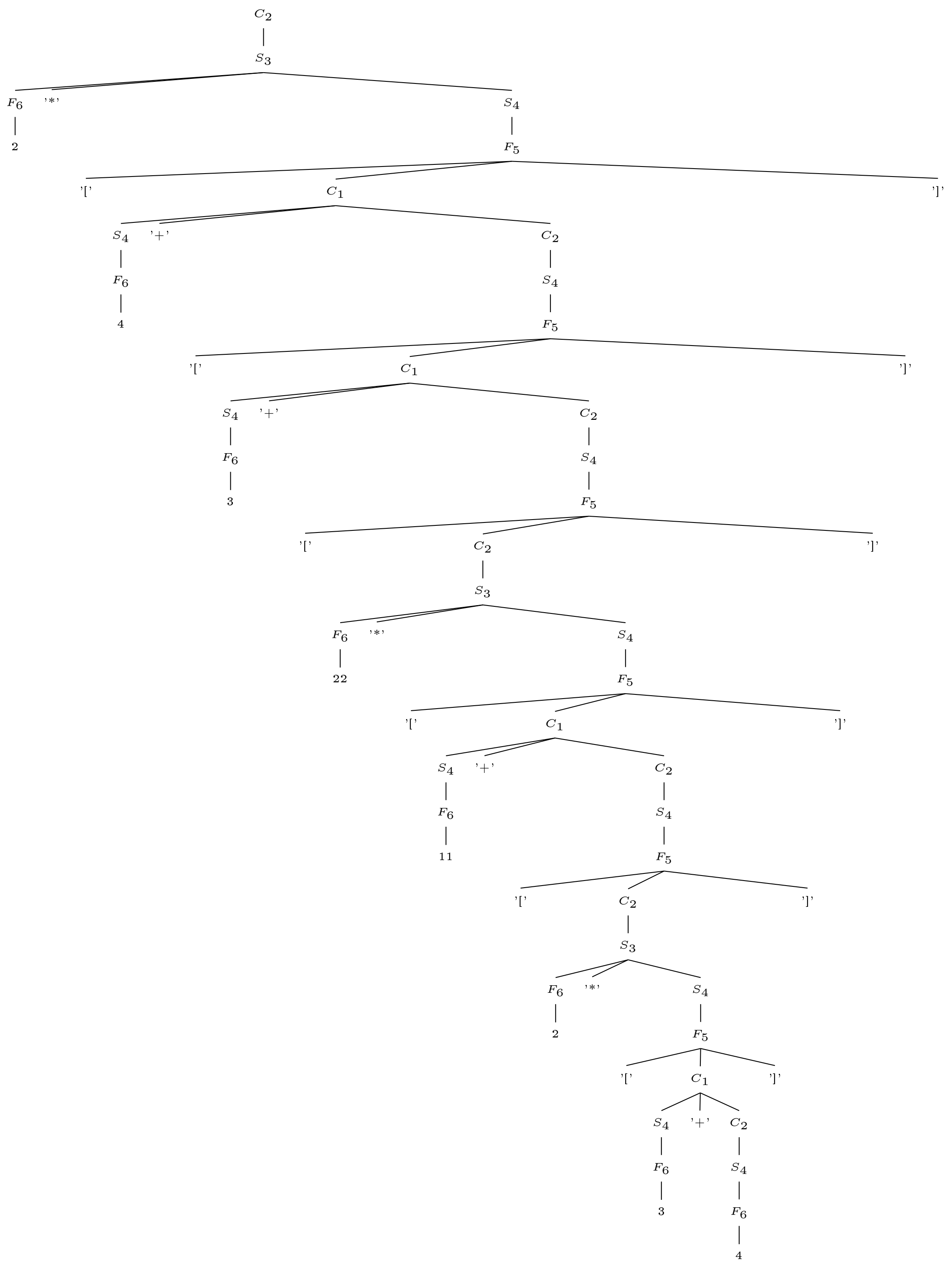
```
start A
begin
  W = A
   begin
     W_1 = collapse W C_1
     in
       Y -> Y_1 Z:1 x:2 Y:3 | Y_2 Z
     out
       W -> W_1 W:1 x:2 W:3
     pattern auto
       Y_2 x     = (x :: W)
   end
  begin
    in
      S -> S_1 x B y | S_2 z:1
    out
      W -> W_2 z:1
    pattern auto
      S_1 x y z = (y :: W)
  end
end
```

The now missing pattern synonyms are inferred from the information flow. the recursing patterns still have to be written by hand since, as mentioned before, to write down the information flow you would have to write infinitely many recursions to cover all depths. For accessing even more complicated parts of the syntax tree, see the next case study.
If the information flow is somewhat linear, you can write **seq** instead of **out**. This will sequentially number all non-assigned matchers; therefore you can specify the odd ones yourself and let the rest be done automatically. In the example, this can be used in the first matcher/producer block, since all information flow is sequentially. Note that Y_2 Z will be numbered as well to Y_2 Z:4, however this does not matter since the information in 4 can't flow to anywhere and so won't produce any pattern. The final file looks like this:

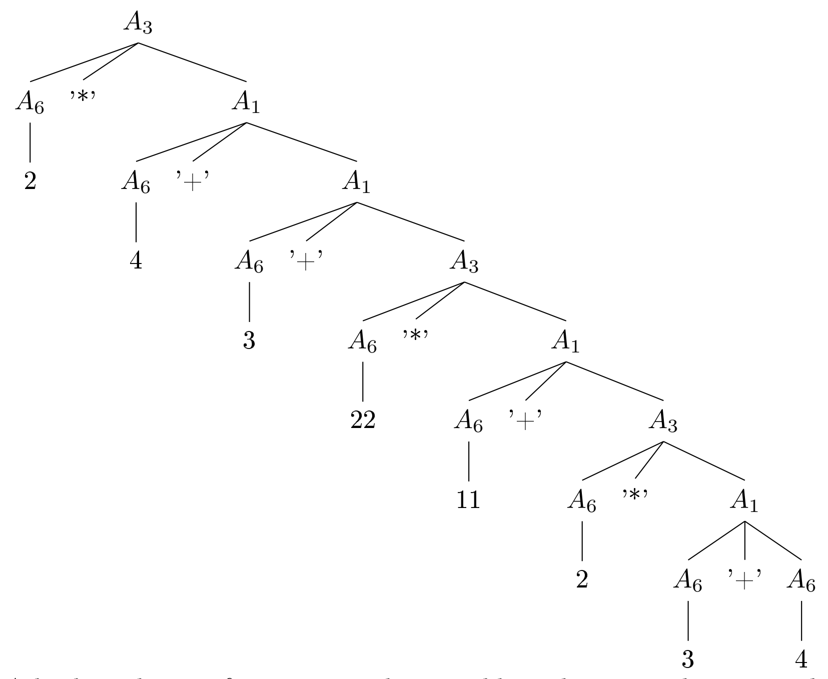
```
start A
begin
  W = A
  begin
    W_1 = collapse W C_1
    in
      Y -> Y_1 Z x Y | Y_2 Z
    seq
      W -> W_1 W x W
    pattern auto
      Y_2 x       = (x :: W)
  end
  begin
    in
      S -> S_1 x B y | S_2 z:1
    out
      W -> W_2 z:1
    pattern auto
      S_1 x y z = (y :: W)
  end
end
```

3.2.1 Sample transformation

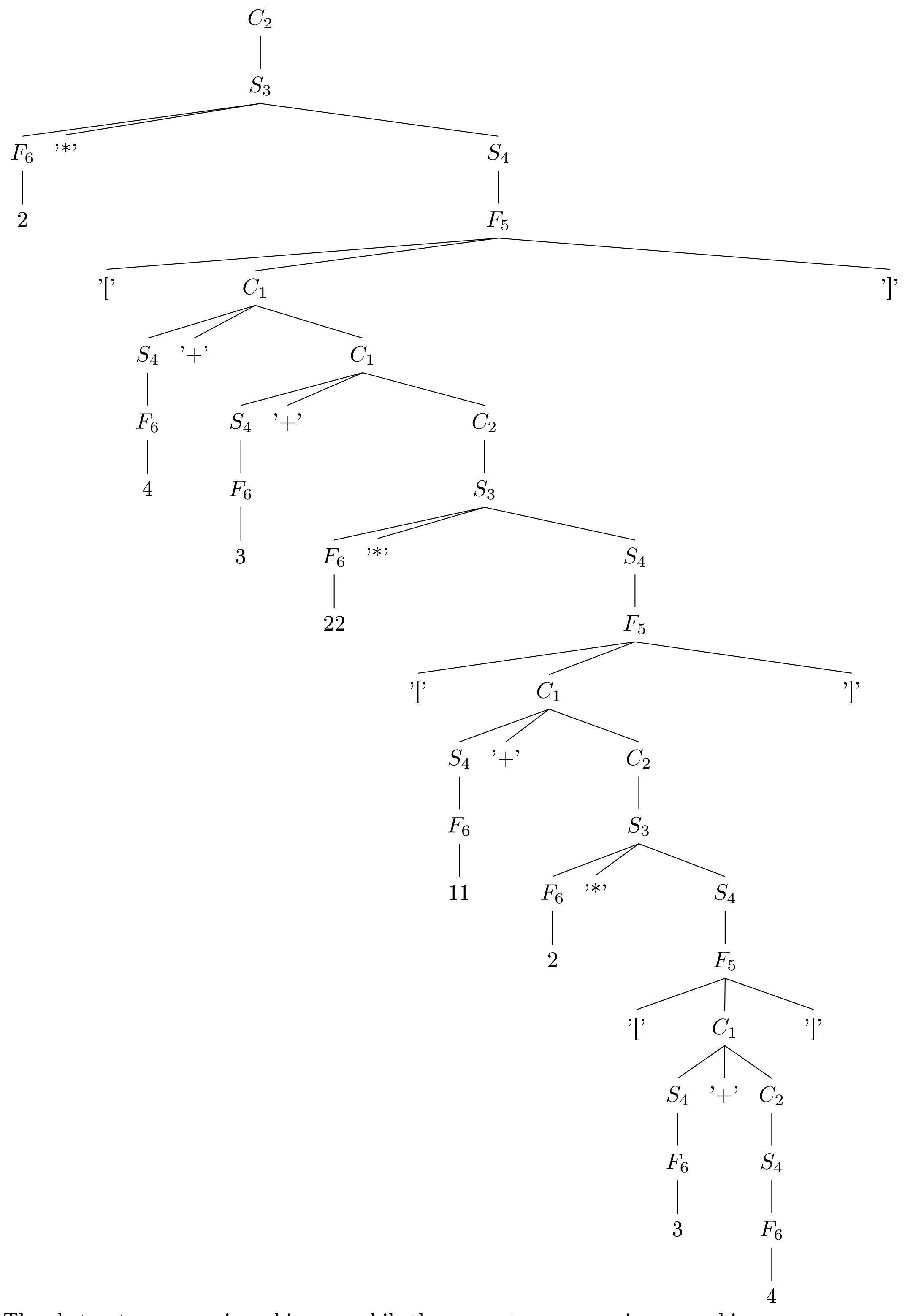
Let's look at an actual conversion. We parse `2*[4+[3+[22*[11+[2*[3+4]]]]]]` using the concrete grammar and get:



After the automatic transformation we get the following syntax tree belonging to the abstract grammar:



A backwards transformation is also possible and generated. We get the following tree:



The abstract grammar is ambiguous while the concrete grammar is non-ambiguous.

Transforming the tree forwards and backwards keeps the precedence of the concrete grammar and it eliminates (or adds) unneeded braces. Obviously there's infinitely many possible ways to insert braces in the tree and All versions of valid backtransformed trees are obtainable; for convenience the least deep, shortest unparsed is choosen for printing. In our case this means that tranforming forwards and backwards eliminates unnecessary braces. The forward transformation only produces one tree (in this case study).

3.3 Transformer-instructions file

To easily use the program, there is the so-called transformer-instructions file. With it, you can specify

- what grammar to load using `"FileName.grammar"`. This has to be the first instruction of the file.
- what transformations to apply using `trans("FileName.tr")` or `exhst("FileName.tr")`. The first applies the transformer file once, the second one applies ist exhaustively, i.e. until nothing changes by applying it again.
- what to parse and with which grammar: `gTran("expr")` parses using the transformed grammar and then transforms the syntax tree back to one of the original grammar, `gOrig("expr")` does the opposite.
- to save the transformed grammar using `writeGrammar("FileName.grammar")`.

You can chain these commands to save intermediate results or apply multiple transformations. Applying two transformations after another chains them, so the resulting grammar will be

$$g_1 \to t_2(t_1(g_1))$$

4 Inner workings

In this section I'll discuss how the algorithm implemented executes transformations.

4.1 .tr to syntax tree transformations

The transformation process consists of the following steps:

1. Matching grammar rules:
   (a) Compare atom by atom
   (b) Keep track of new matches in three different tables:
      i. The SymbolTable keeps track of exact matches, i.e. MatcherAtom `S` matches GrammarAtom `A` or `"xyz"`
      ii. The RuleNameTable keeps track of which rule-matcher matches which grammar rule
      iii. The NameTable keeps track only what the matched rules' names were
2. Producing grammar rules:

   (a) for each matched block of rules, produce grammar rules atom by atom, looking up MatcherAtoms in the SymbolTable and guess them if not yet existing

3. From pattern-synonym prototypes and information flow, produce pattern-synonyms
4. Translate pattern-synonyms to Prolog-definitions

4.2 Different atomic and composite types

- GrammarAtoms are used to represent the right side of grammar rules. There are the following subatoms:
  - Nonterminals
  - TerminalLike: Terminals ("abc"), Regexs ("[0-9a-z]".r), IntegerTerminals and FloatTerminals (<int>/ <float>)
  - GrammarAtomSequence: used to capture matched ...s, can be used in parsing.
- TransformerAtoms are used to represent everything on the right side of matcher and production rules. There's the following subatoms:
  - AnyMatcher matches any GrammarAtom
  - NonterminalMatcher, TerminalMatcher match anything of their respective type
  - LiteralMatcher matches all Terminals with the exact same string
  - IntegerMatcher and FloatMatcher match IntegerTerminals and FloatTerminal s
  - RestMatcher is used to associate with a GrammarAtomSequence. It is used before matching atom by atom.
- PatternAtoms make up the contents of both sides of a pattern-synonym:
  - PatternLiterals, PatternTerminals, PatternIntegers and PatternFloat s are analogous to TransformerAtoms.
  - TypedPatternVariables correspond to Nonterminals and NonterminalMatcher s. After instantiating they are what determines different type errors.
  - PatternAtomPrototypes are generated by the parser when parsing `.tr`-files. They later get instantiated to all other pattern atoms after rules have been matched and produced.
  - ExtractorPattern is a special atom generated by the parser to represent a "loosely typed" pattern side, i.e. the left hand side in (x :: S) =R_1 t x. They are modified slightly when instantiating PatternAtomPrototypes.
  - TypedPattern holds a sequence of PatternAtoms. It is a PatternAtom itself to model pattern synonyms like S_1 (S1_1 x y)t z =S1_1 x (S_1 y t z).

4.3 Applying a transformer block

To apply a transformer block, that is a set of in and out rules and pattern synonym prototypes, we have to do the following:

- Match every in-rule to a grammar-rule. For this, we select every subset of the same size out of the grammar's rules and try matching rule-by-rule, atom-by-atom. This aproach was choosen since we wouldn't know if the first successful match was the one intended if we'd stop after it.
- For every match found this way we get a symbol table, rule-name-table and a name table. With each of these three, we produce a grammar-rule for every out-rule.

This all is done in applyRule.
It get's called by applyMatcherAndTransformer, which then

- collects these rules and patterns
- calls producePatternSynonyms, which does automatic pattern generation
- instantiates pattern-synonym-prototypes
- translates pattern-synonyms to Prolog definitions

4.3.1 Matching rules

Matching one grammar rule to a matcher is done in matches. After putting everything from a grammar rule longer than a matcher inside a rest matcher (if available), it goes through every pair of grammar and matcher atoms and checks the symbol table, if it should continue:

- If the matcher atom is already in the symbol table, continue if the corresponding grammar atom is the same
- If the matcher atom is not found in the symbol table, add the pair of atoms to it and continue
- Otherwise matching fails in this run

After matching all atoms, it returns the gathered information: the modified symbol table. The symbol table also holds the restMatcher/GrammarAtomSequence match.

4.3.2 Producing rules

Producing a grammar rule is simpler than matching one: Go through all matcher atoms and in most cases the symbol table already tells us everything we need to know or we just need to take literal information out of a matcher. Only when a new nonterminal is introduced we have to be creative. For this, there is a Set of used symbols, which we check before adding a new nonterminal.

4.4 Pattern synonyms

4.4.1 Instantiating pattern synonym prototypes

Instantiating the pattern synonyms basically means recursively going through them and replacing all matcher types with the mapped grammar types according to the symbol table. This is done by finalizePattern .

4.4.2 Automatic generation

Patterns can be automatically inferred from information flow, this is handled by producePatternSynonyms. To do this it basically tries to infer some order of derivation of grammar rules, so that every part of information is present on both sides. Example:

```
in
  C -> C_1 S:1  x:2  C:3
out
  A -> A_1 A:1 B
  B -> B_1 x:2  A:3
     | B_2 x:2
```

This will expand into the following pattern synonyms one after another (and some others, which fail):

```
C_1 s x c = A_1 s ?
C_1 s x c = A_1 s (B_1 x c)
```

The question mark here denotes a variable with no associated information flow. The following expansion also happens but fails, since the variable `c` is not resolved:

```
C_1 s x c = A_1 s ?
C_1 s x c = A_1 s (B_2 x)
```

4.5 Prolog

Prolog handles the syntax tree transformation. To talk to Prolog, there's a supplied interface between Prolog and Java, and since Scala compiles to the JVM, we can use it. However it behaves strangely in some cases, which stops us from getting different transformed trees as a stream and other nice-to-have features.
The transformation is done as follows:

- load the previously from pattern synonyms translated definitions
- translate the syntax tree
- query Prolog
- translate the answer(s)

4.5.1 Pattern synonyms to Prolog definitions

Every pattern corresponds to some Prolog definition like

```
relT1toT2(cA_1(Vc1, Vc2, ...), cA_2(...)) :-
  relS1toS2(...),
  ...
```

We'll go through the process of translation for three synonyms to show different type errors. Type errors tell us when Prolog needs to descend into a relation.

Straight translation
Pattern synonym:

```
S_2 (x :: F) = A_1 (x :: F) (R_2 "")
```

This first pattern describes the relation between S_2 and A_1: rename the node, put an empty string in the new leaf. The type of the first leaf stays the same, therefore the corresponding Prolog definition is simple:

```
relStoA(cS_2(VxF), cA_1(VxF, cR_2(''))).
```

In Prolog relations always start with a lowercase letter, variables always start with an uppercase one. The algorithm denotes relations between types S and T like relStoT, constructors (=rule names) like cNT_Name and variables like VinflowTYPE. Strings in Prolog are delimited by single quotes.

Type error between variables
Pattern synonym:

```
 (x :: S) = R_1 "+" (x :: A)
```

As you can see, the variable `x` is not of the same type on the left and right hand side. We therefore have to tell Prolog, that it should try to relate between them by adding a constraint on this variable:

```
relStoR(VxS, cR_1('+', VxA)) :-
  relStoA(VxS, VxA).
```

Type error because of missing constructor
Pattern synonym:

```
  S_1 (A_1 (x :: F) (y :: R)) "+" (z :: F)
= A_1 (x :: F) (S_1 (y :: S) "+" (z :: F))
```

In this pattern synonym the constructor for A_1 occurs on the left hand side, even though A is a nonterminal of the right hand side's grammar. We therefore have to translate the content of the first leaf of S_1, which is of type R, to the type A. The same holds for the right side. Additionally, we encounter a type error on `y`. Both of these resolved yield the following Prolog code:

```
relStoA(cS_1(RA_1, '+', VzF), cA_1(VxF, RS_1)) :-
  relStoR(VyS, VyR),
  relStoA(RA_1, cA_1(VxF, VyR)),
  relStoR(cS_1(VyS, '+', VzF), RS_1).
```

4.5.2 Loading definitions

The object PrologInterface comes with a method transformGrammarWithFile, which takes a grammar and the path to a file containing a transformer object. It returns the transformed grammar and two methods to convert between the grammars. These methods hold the definitions generated like above and on call load the definitions and translate the trees like below. To load the definitions, the methods write them to a temporary file that is loaded and, after transformation, is unloaded and deleted again.

4.5.3 Syntax tree translation

Syntax trees relate to Prolog terms quite easily:

- Branches correspond to Compounds with a constructor like cNT_Name. The subbranches correspond to the Compound's content and are translated recursively.
- LeafStrings correspond to Atoms
- LeafIntegers and LeafFloats correspond to their respective Prolog equivalents Integer and Float

After transformation, Prolog returns a Term and a Map from String to Term which resolves variables. The backwards translation is therefore equally easy and contains only a bit of string magic.

5 Discussion

5.1 Time complexity

If you've read the above carefully, you'll notice it says a lot of things like "every subset", you might worry about exponential running time. While this is true, it's not relevant at this stage, since:

- Grammars and transformers are relatively small compared to the grammars' syntax trees
- Grammars are transformed once, then multiple trees are transformed (usually)
- Prolog is way slower, since it explores all transformation paths with the same diligence as we explore all matches

So yes, runtime is exponential in the size of the grammar and the size of the transformer blocks, but no, it's not that bad, since there is another part in this which is the bigger performance sink. Some speedup in that area therefore would be nice, either in another aproach than the Prolog one or in some guidelines for Prolog how to explore the relations. However this is not part of this thesis' objective.

5.2 Problems

In the *.tr*-file, the first line hard-codes the transformed grammar's start symbol. This is unconvenient, but a neccessary evil at this point. We'd like to specify the start symbol depending on the input grammar, specifically when matching and producing rules. However when specifing that the start symbol should be taken from some match of a matcher/producer pair, you won't know from which match in case of multiple matches, in case of no matches you wouldn't know which symbol to take at all, ...
Moreover the matching/producing approach does not define a relationship between the atoms of the input and output grammar, therefore you can't say you'd like to just transform the start symbol like you transformed the rules.
There's multiple solutions I can think of, which all have their drawbacks:

- specify the start symbol in the `.ti`-file instead of the `.tr`-file

  ↯ while this makes the start symbol independent of the transformation, it only moves the problem but does not solve it. By putting the start symbol in the `.tr`-file at least you can hard code the produced grammar's rules' left hand sides as well.

- specify the start symbol according to some match, use only the last/first/... match, default to A.

  ↯ This may work in many cases, but it's also difficult to think around and prevents any kind of multitasking when matching/producing.

- specify no start symbol.

  ↯ Nope. You wouldn't even be able to define the transformation relation.

6 Case studies

In this section we'll look at some example transformations to show the potential and limitations of the current algorithm.

6.1 Eliminating left-recursion

6.1.1 Grammar

```
start S
S -> S_1   S "+" F   | S_2 F
F -> F_3 "[" S "]" | F_4 <int>
```

Eliminating left-recursion is quite a difficult task. We therefore start with a simplified left-recursive version of the concrete grammar.
At first, you might be tempted to want to transform this grammar into one like so:

```
start C1
begin
  begin
    in
      S -> S_1 S t F | S_2 F
    seq
      S -> S_3 F t S | S_4 F
    pattern auto
  end
  // ...
end
```

Although the resulting grammar is indeed not left-recursive, the transformation fails, since you can't put `S`es into `F`s and vice versa. You might therefore consider this transformation:

```
start C1
begin
  begin
    in
      S -> S_1 S:1 t F:2 | S_2 F
      seq
        S -> S_3 F:2 t S:1 | S_4 F
      pattern auto
    end
    // ...
end
```

Even though this defines a valid transformation, it does not preserve the shape of the tree: instead it reverses it. What we really want is to turn the whole tree like a wheel. This is achieved using the following transformer:
(the rule was split up to look more like the textbook example of eliminating left-recursion)

```
start A
begin
  S1 = A
  begin
    S1_1 = collapse S1 S_1
    in
      S -> S_1 S t F | S_2 F
    out
      S1 -> S1_1 F R
      R  -> R_2 "" | R_1 t S1
    pattern
      S_2 x = S1_1 x (R_2 "")
      S_1 (S_2 x) t y  = S1_1 x (S_2 y)
      S_1 (S1_1 x y) t z = S1_1 x (S_1 y t z)
      (x :: S) = R_1 t x
  end
  //this part is straight forward information copying
  begin
    in
      A -> A_2 x B y
    seq
      A -> A_2 x S1 y
    pattern auto
  end
  begin
    S1 = F
    S1_1 = collapse S1 A_1
    in
      A -> A_1 <int>
    seq
      S1 -> S1_1 <int>
    pattern auto
  end
end
```

6.1.2 Transformed grammar

```
start A
```

```
F -> F_3 "[" A "]"
   | F_4 <int>
A -> A_1 F R
R -> R_1 "+" A
   | R_2 ""
```

Each of the pattern synonyms will now be analysed on it's own. I'll use the instantiated ones, since they are more verbose and type annotated. As a reminder: the left-hand side corresponds loosely to the input grammar, the right-hand side to the transformed grammar.
`(S_2 (x :: F)) = (A_1 (x :: F) (R_2 "")):`
This is the easiest pattern synonym: We encountered a S_2-branch containing an F. This is easily stored in an A_1 with no rest R.

`(S_1 (S_2 (x :: F)) (t :: "+") (y :: F)) = (A_1 (x :: F) (S_2 (y :: F))):`
The variable `x` again is easy: we are on one of the last left-recursive branches, so we can put the contained information directly into a new A-branch.
`y` is slightly more complicated: It looks easy enough on the left hand side: it's direct information in our branch. On the right hand side it is written as an S_2 though, which is strange at first, since S_2 belongs to the input grammar. In that case, when the written type and the expected type does not fit, i.e. we've gotten a type error, the algorithm tries to transform this not-matching pattern synonym branch to the correct type. In short, this subpattern means: take the `y`, wrap it in a S_2 branch, and try to go from there.
It's not needed in this case, but it demonstrates a neccessary and powerful feature.

```
  (S_1 (A_1 (x :: F) (y :: R)) (t :: "+") (z :: F))
= (A_1 (x :: F) (S_1 (y :: S) (t :: "+") (z :: F)))
```

This is the main pattern of the conversion: it expresses rotating the entire tree. The left hand side extracts the left-most leaf `x` and corresponds to what we want to get at the end, since the constructor A_1 will be the left-most subtree. `y` contains everything between `x` and the right side of our tree, `z`.
The right hand side stores this `x` directly, storing the right-most leaf is done again by putting it in a rule that can store the middle of the tree and the right-most leaf.
This rule raises three type errors: The inclusion of A and S in one-another are already familiar; the type error arising from `y` beeing either an R or an S is new. All type errors are dealt with by the algorithm, note that you can use this to do subtree conversions.

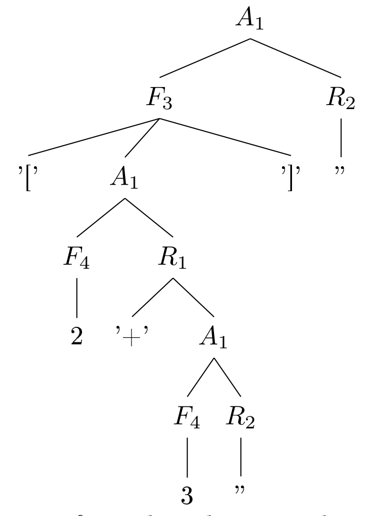
`(x :: S) = (R_1 (t :: "+") (x :: A)):`
This pattern is both trivial and interesting: It just stores some `x` of type S in an R-rule. It's interesting because it demonstrates a feature: If you don't care about the constructor of some branch but know how to transform it to a branch of another type like we do with S to A because of the first three patterns, you can generalize the constructors into a type-annotated variable instead.
Note, that this is the first pattern not relating types S and A, but S and R. The algorithm is not confused by this since it chooses from the applied patterns by type relations.

The following two patterns are autogenerated copy patterns.

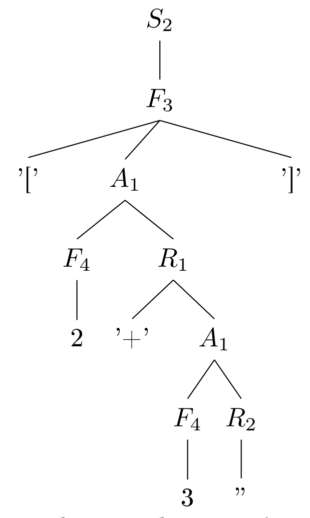
```
(F_3 (1 :: "[") (2 :: S) (3 :: "]")) = (F_3 (1 :: "[") (2 :: A) (3 :: "]"))
(F_4 (1 :: <int>)) = (F_4 (1 :: <int>))
```

6.1.3 Sample transformation

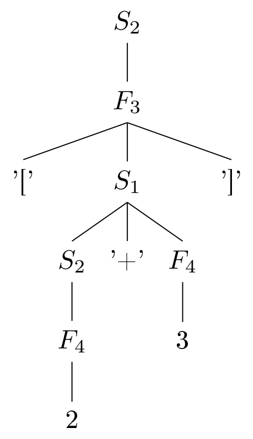
We start by parsing $[2 + 3]$ with the resulting grammar:



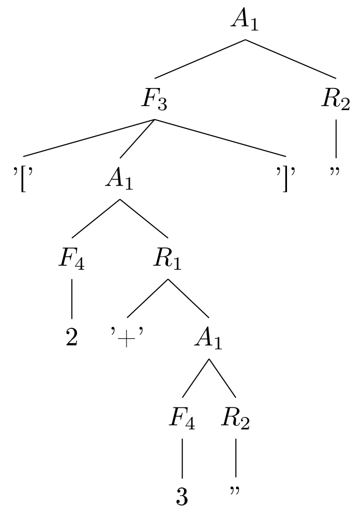
Transformed to the original grammar we get:



Woah, something wen't wrong there! The tree still contains subtrees of type A and is therefore not of the original grammar. This happened, because we didn't change the type F, so the translation of pattern synonyms assumed we were done. To enforce deeper recursion, we add the keyword `force` to `pattern`. This increases tree translation time by a lot, but yields the correct backwards tree:



Transformed to the right recursive grammar again:



6.1.4 More complex grammars

Transforming more complex grammars and their syntax trees, as in the first case study, is possible but so slow that trees of a depth of more than eight take more than 20 minutes and 12GB RAM without finding a solution on a recent, fast CPU. This is at least partly due to prolog doing more in-depth searches than possibly needed and not searching multi-threaded and could be something to work on in the future.

6.2 Left-factoring, inlining and Chomsky-two-form

Left-factoring, inlining and Chomsky-two-form are all quite easy, since they all only insert or remove Nonterminals and move stuff around. As an example we'll look at chomsky-two-form in detail and only show example transformations of the other two. Chomsky-two-form is a variant of chomsky-normal-form where we don't require all rules to be of one of the following forms:

```
A -> B C
A -> D
```

```
A -> <TERMINAL>
```

but rather only require them to have at most two atoms on the right hand side.

6.2.1 Grammar

```
start S
S -> S_2 "if " E " then " S " else " S
    | S_1 "if " E " then " S
    | S_t <int>
E -> E_T "1" | E_F "0"
```

We would like to get some kind of grammar like this, an equivalent grammar in Chomsky-two-form:

```
start S
S -> S_2 S1 S
    | S_1 S5 S
    | S_t <int>
E -> E_T "1" | E_F "0"
S1 -> S1_1 S2 " else "
S2 -> S2_1 S3 S
S3 -> S3_1 S4 " then "
S4 -> S4_1 "if " E
S5 -> S5_1 S6 " then "
S6 -> S6_1 "if " E
```

6.2.2 Transformer

```
start S
begin
  begin
    A1 = newName A1
    in
      A -> A_1 _B:1 _C:1 ...D
    out
      A  -> A_1 _B:1 A1
      A1 -> A1_1 _C:1 ...D
    pattern auto force
  end
  begin
    in
      C -> C_1 _A _B
    seq
      C -> C_1 _A _B
    pattern auto force
  end
  begin
    in
      C -> C_1 _A
    seq
      C -> C_1 _A
      pattern auto force
    end
end
```

As you can see here, many things can be done automatically and it's easy to write an understandable transformer file. Note the `...D`-atom: it matches one or more nonterminals and terminals. Also note the `force` toggle: it makes the algorithm try harder to transform subtrees. This is needed whenever a subtree's content should change, but it's type stays the same, which is mostly needed when doing exhaustive transformations since you have to keep types unchanged when copying rules, otherwise the algorithm doesn't know when to stop applying rules.
Applying this file only once doesn't fully transform the grammar but yields the following:

```
start S
A11 -> A11_1 E " then " S

S -> S_2 "if " A10
   | S_1 "if " A11
   | S_t <int>

E -> E_T "1"
   | E_F "0"

A10 -> A10_2 E " then " S " else " S
```

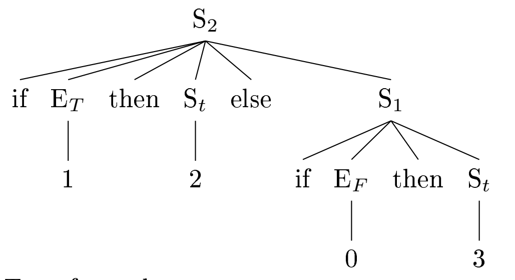
What really is necessary is to apply the transformation exhaustively, i.e. until nothing changes anymore. Therefore in the transformer-instructions-file, we write `exhst("chomsky.tr")` instead of `trans("chomsky.tr")`. This produces:

```
start S
A15 -> A15_2 " else " S

E -> E_T "1"
   | E_F "0"

A11 -> A11_1 E A13

S -> S_2 "if " A10
   | S_1 "if " A11
   | S_t <int>

A12 -> A12_2 " then " A14
A13 -> A13_1 " then " S
A14 -> A14_2 S A15
A10 -> A10_2 E A12
```

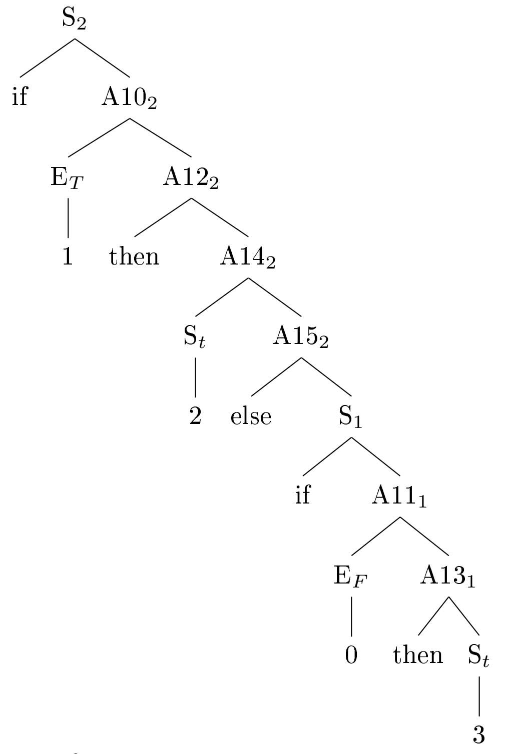
which indeed is of the chomsky-two-form.

6.2.3 Sample transformation

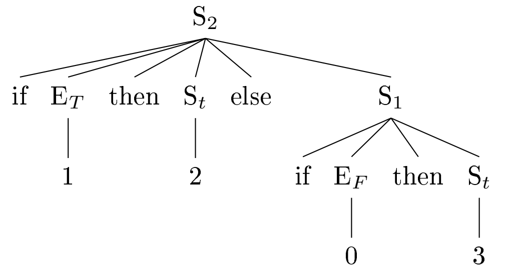
We'll start by parsing `if 1 then 2 else if 0 then 3`:



Transformed:



Transformed backwards:



6.3 Left-factoring

In this case study we want to factor out some parts of a grammar rule, so that different rules with the same nonterminal on the left hand side only contain up to one common atom on the right hand side.

6.3.1 Grammar

```
start S
S -> S_2 "if " E " then " S " else " S
    | S_1 "if " E " then " S
    | S_t <int>
E -> E_T "1"
    | E_F "0"
```

6.3.2 Transformed grammar

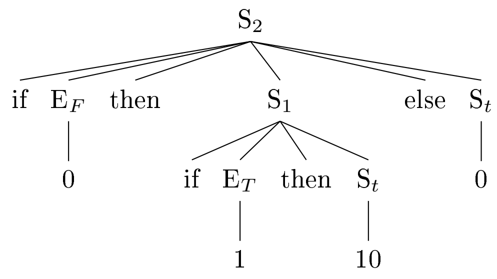
```
start A
E1 -> E1_T "1"
     | E1_F "0"
A -> A_1 "if " E1 " then " A S2
    | A_t <int>
S2 -> S2_1 " else " A
     | S2_2 ""
```

6.3.3 Transformer

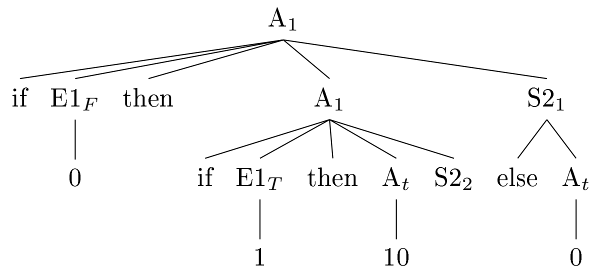
```
start A
begin
  A = A
  A_1 = collapse A S_1
  in
    S -> S_1 tif E:1 tthen S:2
       | S_2 tif E:1 tthen S:2 te S:3
       | S_3 tt:4
  out
    A  -> A_1 tif E1:1 tthen A:2 S2 | A_3 tt:4
    S2 -> S2_1 te A:3 | S2_2 ""
  pattern auto
end
begin
  E1 = E1
  in
    E -> E_1 x | E_2 y
  seq
    E1 -> E1_1 x | E1_2 y
  pattern auto
end
```

6.3.4 Sample transformation

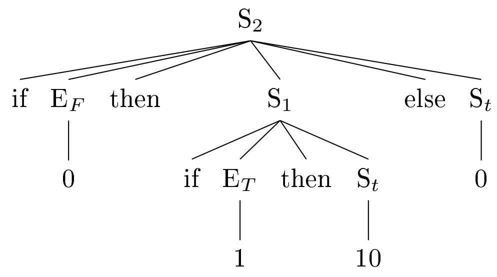
`"if 0 then if 1 then 10 else 0"` parsed:



Transformed:



Transformed backwards:



6.4 Inlining

This case study is effectively the reverse transformation of the last case study: We want to inline some rule to eliminate a nonterminal from the grammar.

6.4.1 Grammar

```
start A
E1 -> E1_T "1"
    | E1_F "0"
A -> A_1 "if " E1 " then " A S2
   | A_t <int>
S2 -> S2_1 " else " A
    | S2_2 ""
```

6.4.2 Transformed grammar

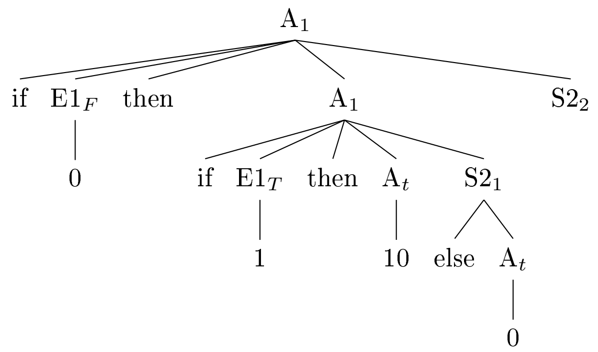
```
start S
S -> S_2 "if " E " then " S " else " S
```

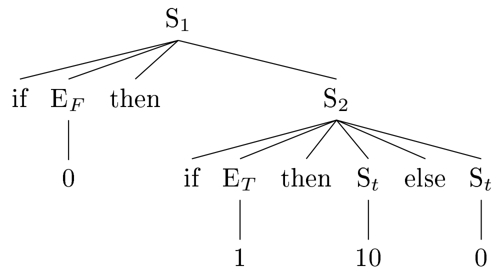
```
    | S_1 "if " E " then " S
    | S_t <int>
E -> E_T "1"
    | E_F "0"
```

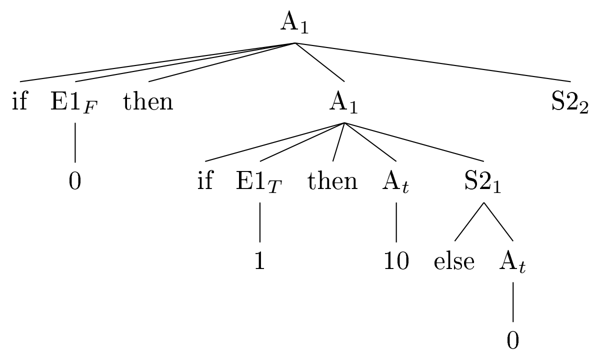
6.4.3 Transformer

```
start S
begin
  S = S
  S_1 = collapse S A_1
  in
    A  -> A_1 tif E1:1 tthen A:2 S2 | A_3 tt:4
    S2 -> S2_1 te A:3 | S2_2 ""
  out
    S -> S_1 tif E:1 tthen S:2
       | S_2 tif E:1 tthen S:2 te S:3
       | S_3 tt:4
  pattern auto
end
begin
  E = E
  in
    E1 -> E1_1 x | E1_2 y
  seq
    E -> E_1 x | E_2 y
  pattern auto
end
```

6.4.4 Sample transformation







7 Related works

7.1 Generating attribute grammar-based bidirectional transformations from rewrite rules

This work describes a process to generate bidirectional syntax tree transformations from attribute grammar rewrite rules[2]. These rewrite rules are similar to our pattern synonyms, although directed (i.e. always from left to right) but not easily invertible and more constrained in the following ways:

- The left hand side has to be unique disregarding values of parameters, i.e. the following corresponding pattern synonyms would all be considered illegal duplicates of one another:
  - `A_1 x =...`
  - `A_1 y =...`
  - `A_1 (A_2 x)=...`

  These restrictions are placed to make the transformation a function, with some workaround to the last case appearing together with one of the first two introduced later.
- Recursion into the left hand side is not possible, making left-recusion elimination impossible, since you need an infinitely finite level of recursion to get to the left-/right-most child.

Our work doesn't try to generate functions as transformations but instead generates relations, therefore allowing more than one transformation result and allowing overlapping patterns.
The paper also doesn't cover grammar transformations.

7.2 XSLT

XSLT (XSL Transformations) are a way to describe how to transform XML-files (eXtensible Markup Language) to some other document [3]. To describe how to transform a XML, you have access to

- searching for a tree node by it's type/node name, or some pattern matching it or it's relative or absolute position in the tree
- looping through children
- state, as in variables

- conditional statements

Using these, you can extract information from the XML-nodes and put them into any target non-/structure.
Since every syntax tree can be transformed to some XML quite easily, this can be seen as an approach to modifying parser results, without

- grammar transformation
- backwards transformation
- (easy) reuse of transformations for different grammars

It therefore doesn't solve left-recursion elimination for recursive descent parsing (it can however describe how to reverse the transformation). XSLT support is implemented in all major browsers [4].

7.3 biXid: a bidirectional transformation language for XML

This work explores bidirectional XML transformations using relations and XML-tree walking[5]. It describes relations in a similar way to our Prolog definitions. The following example transforms a Netscape bookmark to an XBEL bookmark:

```
relation top =
  html[head[String],
    body[h1[var t as String], dl[var nc]]]
<->
  xbel[title[var t as String], var xc]
where
  contents(nc, xc)
```

The corresponding Prolog definition would be:

```
relHTMLtoXBELtop(
  chtml_1(chead_1(V1S), cbody_1(ch1_1(VtS), cdl_1(Vnc))),
  cxbel_1(ctitle_1(VtS), Vxc)
) :- relHTMLtoXBELcontents(Vnc, Vxc).
```

However they place some restrictions on the relations which forbid many transformations to transform the XML using two regular automata, no OR in the where-clause is the most significant one. Rotating a tree is not possible, since that needs that exact recursion method, and so transforming between left- and right-recursive trees is impossible, the hardest transformation we looked at. Also, transforming grammars is not covered.

8 Conclusion

We tried to come up with a way to describe transformations between (context-free) grammars and generate corresponding syntax-tree transformations. Throughout this work we've found that it's not enough to describe how grammar rules change into each other, it's often needed to describe the syntax tree conversions as well.
In some simple cases, describing the information flow may be enough, in other cases pattern synonyms can be enough to express the transformations. It's not clear at this point if there is a need to extend the way of describing the syntax tree conversion further, for example by writing prototype Prolog code.

References

[1] Martin Lange, Hans Leiß, *To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm*, 2009.

[2] Martins, Saraiva, Fernandes, Wyk., *Generating attribute grammar-based bidirectional transformations from rewrite rules*, 2014.

[3] Michael Kay, Saxonica, W3C, *XSL Transformations (XSLT) Version 2.0*, 2007. `http://www.w3.org/TR/xslt20/`

[4] w3schools, *XSLT Introduction*, 2015. `http://www.w3schools.com/xsl/xsl_intro.asp`

[5] Kawanaka, Shinya, and Haruo Hosoya, *biXid: a bidirectional transformation language for XML*, ACM SIGPLAN Notices. Vol. 41. No. 9. ACM, 2006.

`http://en.wikipedia.org/wiki/Formal_grammar`
Source code: `https://github.com/plneappl/bsc-thesis`

Selbständigkeitserklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Simon Wegendt

Tübingen, den 20. Oktober 2015